AJAY KADIYALA - Data Engineer

Follow me Here:

LinkedIn: https://www.linkedin.com/in/ajay026/

Data Geeks Community:

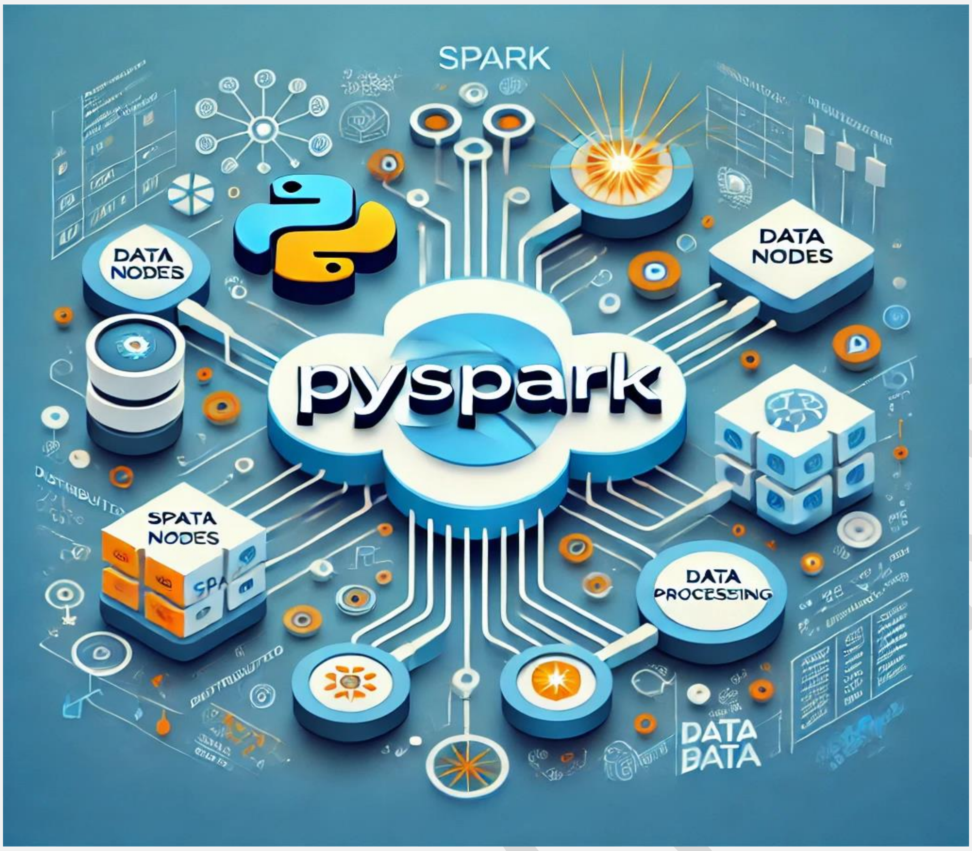
https://lnkd.in/gU5NkCqi

PYSPARK

DATA ENGINEER INTERVIEW

QUESTIONS & ANSWERS

What is Pyspark??



PySpark is the Python API for Apache Spark, which is one of the most popular frameworks for big data processing. It allows data engineers to handle massive datasets across distributed clusters, enabling parallel processing and faster computation. Unlike traditional Python, which is limited to processing data on a single machine, PySpark leverages Spark's power to distribute tasks across multiple nodes, making it incredibly efficient for large-scale data transformations and ETL (Extract, Transform, Load) workflows.

Why Pyspark is Important for Data Engineer?

In the context of data engineering, PySpark is crucial because it bridges the gap between the flexibility of Python and the scalability of Spark. For instance, data engineers often work with vast amounts of raw data, where single-machine processing becomes impractical. PySpark not only handles large datasets but also offers optimized transformations and actions that save both time and resources.

For interviews, PySpark knowledge is increasingly essential because many data engineering roles involve big data infrastructure. Interviewers are likely to test candidates on their ability to perform data wrangling, implement distributed algorithms, and optimize pipeline performance using PySpark.

Table of Contents

1. What is PySpark, and how is it different from traditional Python?

- **Answer**: PySpark is the Python API for Apache Spark, which allows us to leverage Spark's distributed computing capabilities within Python. Unlike traditional Python, which runs code sequentially on a single node, PySpark enables parallel processing over a cluster, making it highly efficient for big data. PySpark is especially useful for tasks that require distributed data processing, such as ETL, machine learning pipelines, and large-scale data transformations.

2. Can you explain the difference between RDDs, DataFrames, and Datasets in PySpark?

- **Answer**: Sure. RDD (Resilient Distributed Dataset) is the fundamental data structure in Spark, designed for low-level transformations and actions. It's untyped and can handle unstructured data. DataFrames, on the other hand, are more like tables with named columns, optimized for structured data. They support SQL-like operations and provide better performance due to Catalyst optimizer. Datasets combine the benefits of RDDs and DataFrames, offering type safety and the performance advantages of DataFrames, but in PySpark, we mostly use DataFrames because Datasets are primarily available in Scala.

3. How does lazy evaluation work in PySpark?

- **Answer**: Lazy evaluation in PySpark means that Spark doesn't execute operations as soon as they're called. Instead, it builds a logical plan, waiting until an action (like collect(), count(), etc.) triggers execution. This allows Spark to optimize the execution plan by combining transformations, which can significantly reduce the number of operations and optimize the workflow before actually running it.

4. Explain the concept of a DAG (Directed Acyclic Graph) in Spark.

- **Answer**: In Spark, a DAG represents the sequence of transformations and actions on data, organized in stages. Each node in the DAG represents an RDD, and each edge represents a transformation (like map, filter, etc.). When an action is called, Spark's DAG scheduler breaks the logical plan into stages that can be executed in parallel. This approach improves efficiency by eliminating redundant calculations.

5. How does Spark handle data shuffling, and why is it an expensive operation?

- **Answer**: Data shuffling in Spark is the process of redistributing data across different nodes, usually triggered by operations like groupBy, reduceByKey, and join. Shuffling is costly because it involves disk I/O, network I/O, and serialization. Spark minimizes shuffling with transformations that use combiner functions, like reduceByKey, which allows data aggregation before shuffling.

6. What are some common transformations in PySpark?

- **Answer**: Common transformations include map, flatMap, filter, distinct, union, groupByKey, and reduceByKey. These transformations create new RDDs from existing ones and are "lazy," meaning they are only executed when an action is triggered.

7. Can you explain the difference between map() and flatMap() in PySpark?

- **Answer**: Yes, map() applies a function to each element of the RDD, resulting in a new RDD with the same number of elements as the input. flatMap(), on the other hand, can return multiple values for each input element, which results in a flattened RDD with potentially more elements than the original.

8. How do you perform a join operation in PySpark DataFrames?

- **Answer**: Joining in PySpark DataFrames is done using the join() method. We specify the DataFrames to join, the columns on which to join, and the type of join (inner, left, right, outer). For instance, `df1.join(df2, df1.id == df2.id, "inner")` performs an inner join on the id column.

9. What is the difference between reduceByKey and groupByKey?

- **Answer**: Both reduceByKey and groupByKey aggregate data based on keys, but reduceByKey is more efficient because it combines data locally on each node before shuffling. groupByKey sends all data associated with a key to a single node, which can lead to memory and network issues for large datasets.

10. How would you handle skewed data in Spark?

- **Answer**: Data skew occurs when certain keys have significantly more data, causing some nodes to process more data than others. To handle skew, I could use techniques like salting (adding a random suffix to keys to distribute load), using map-side combine with reduceByKey, or even repartitioning data using repartition or coalesce to balance it more evenly across nodes.

11. Explain what Catalyst Optimizer is and how it works in PySpark.

- **Answer**: Catalyst Optimizer is Spark's query optimization engine for DataFrames and SQL. It analyzes the logical plan of a query and applies various optimization rules to improve performance. Catalyst performs logical optimizations (like predicate pushdown) and physical optimizations (such as selecting the best join strategies) to reduce query execution time.

12. What are broadcast variables, and how do they improve performance in PySpark?

- **Answer**: Broadcast variables allow us to cache a read-only variable on each machine, rather than shipping it with every task. This reduces network overhead and improves performance, especially when working with large reference data that's repeatedly used across tasks.

13. Describe how you would handle null values in PySpark DataFrames.

- **Answer**: Null values can be handled using na functions in PySpark DataFrames. For instance, fill can replace null values with specified values, drop can remove rows containing nulls, and replace can substitute specific values. I typically use these based on the business logic or data requirements.

14. How does PySpark handle schema inference and why is it important?

- **Answer**: PySpark can infer the schema of data when reading from structured data sources like Parquet or CSV. Schema inference is essential for DataFrames because it allows Spark to understand data types, which helps in query optimization and reduces errors related to incorrect data types during processing.

15. What is the purpose of checkpointing in Spark?

- **Answer**: Checkpointing is a way to truncate the lineage of an RDD or DataFrame by saving it to disk. This is useful for long-running jobs where the DAG becomes very large, as it reduces the memory overhead and potential recomputation costs if a failure occurs.

16. How would you use window functions in PySpark, and can you give an example?

- **Answer**: Window functions allow operations across a sliding range of data, such as calculating rolling averages or cumulative sums. In PySpark, I'd define a Window specification and use functions like rank, dense_rank, row_number, lead, and lag. For example:

```
from pyspark.sql.window import Window

from pyspark.sql.functions import row_number


windowSpec = Window.partitionBy("department").orderBy("salary")

df.withColumn("rank", row_number().over(windowSpec)).show()
```

17. Can you explain coalesce and repartition in PySpark? When would you use each?

- **Answer**: repartition changes the number of partitions by shuffling data, making it useful for increasing partitions for parallelism. coalesce reduces the number of partitions without a shuffle, which is efficient when decreasing partitions. I use repartition for wider transformations and coalesce for narrow ones, depending on the operation's needs.

18. How would you optimize a PySpark job to reduce memory consumption?

- **Answer**: I'd focus on reducing shuffles and serialization costs. Using reduceByKey over groupByKey, choosing appropriate data formats (like Parquet), and applying persist() to cache intermediate data are effective. Avoiding UDFs when possible and using built-in Spark functions can also help with memory optimization.

19. How would you set up a PySpark job for real-time streaming data?

- **Answer**: For real-time streaming, I'd use Spark Structured Streaming. First, I'd set up a source (e.g., Kafka) and define a structured query using DataFrame API, applying transformations as needed. Finally, I'd define the sink (e.g., console, file, or Kafka) and start the streaming query with .start().

20. What are accumulators in PySpark, and when would you use them?

- **Answer**: Accumulators are write-only shared variables used for aggregating data across tasks. They're mainly used for counters and debugging, as tasks can update accumulators, but they can't read them. They're useful for tracking metrics like the number of errors or invalid records during processing.

21. Explain the purpose of selectExpr in PySpark DataFrames.

- **Answer**: selectExpr allows us to execute SQL expressions within DataFrames. It's convenient when we need to perform transformations with SQL syntax, such as aliasing, conditional transformations, or performing calculations directly in a single line.

22. How do you handle large file sizes efficiently in PySpark?

- **Answer**: For large files, I usually work with efficient file formats like Parquet or ORC, which support columnar storage and compression. I also tune partition sizes and use distributed storage like HDFS to improve read/write speeds across the cluster.

23. What are UDFs in PySpark, and when should you avoid them?

- **Answer**: User-Defined Functions (UDFs) allow us to write custom functions in Python and apply them to DataFrames. However, UDFs are slower because they operate outside Spark's Catalyst optimizer. I avoid UDFs when possible by using built-in PySpark functions, which are optimized and run faster.

24. How would you debug a PySpark job that is failing due to memory issues?

- **Answer**: First, I'd check the Spark UI to identify stages with high memory usage. Next, I'd look into optimizing transformations, reducing shuffles, and tuning configurations like spark.executor.memory. If necessary, I'd repartition data or persist intermediate results selectively to avoid memory bottlenecks.

25. Can you describe the process of creating an ETL pipeline in PySpark?

- **Answer**: Sure. An ETL pipeline in PySpark involves extracting data from a source (e.g., HDFS, Kafka), transforming it (e.g., filtering, aggregations, joins), and then loading it to a target (e.g., data lake, warehouse). I'd use Spark transformations for the data processing part, along with scheduling tools like Airflow for orchestration.

26. Write a PySpark code to calculate the average salary by department.

```
from pyspark.sql import SparkSession

from pyspark.sql.functions import avg


spark = SparkSession.builder.appName("AvgSalaryByDept").getOrCreate()


df = spark.createDataFrame([

   (1, "HR", 4000),

   (2, "Engineering", 5000),

   (3, "HR", 4500),

   (4, "Engineering", 5500)

], ["employee_id", "department", "salary"])


avg_salary_df = df.groupBy("department").agg(avg("salary").alias("avg_salary"))

avg_salary_df.show()
```

Explanation: Here, we use groupBy on the department column and calculate the average of the salary column using the avg function. This groups the data by each department and provides the average salary for each one.

27. **How would you remove duplicate rows from a DataFrame in PySpark?**

```
df = spark.createDataFrame([

   (1, "John", 1000),
```

```
    (1, "John", 1000),

    (2, "Mary", 2000),

    (3, "Mike", 3000)

], ["id", "name", "salary"])


deduped_df = df.dropDuplicates()

deduped_df.show()
```

Explanation: Using dropDuplicates() removes all rows that have duplicate values across all columns. It's a quick way to clean data of redundancies.

28. **Write code to count the number of null values in each column of a DataFrame.**

```
from pyspark.sql.functions import col, isnan, when, count


df = spark.createDataFrame([

    (1, None, 1000),

    (2, "Mary", None),

    (3, None, 3000)

], ["id", "name", "salary"])


null_counts = df.select([count(when(col(c).isNull() | isnan(col(c)), c)).alias(c) for c in df.columns])

null_counts.show()
```

Explanation: We use a combination of isNull() and when() to check for nulls in each column and count() to sum these occurrences for each column.

29. **How would you find the maximum value in each column of a DataFrame?**

```
from pyspark.sql.functions import max


df = spark.createDataFrame([
```

```
    (1, 1000, 25),
    (2, 2000, 30),
    (3, 3000, 35)
], ["id", "salary", "age"])

max_values = df.agg(*[max(c).alias(c) for c in df.columns if c != "id"])
max_values.show()
```

Explanation: Using agg() with max computes the maximum value for each specified column. This solution excludes columns like id, focusing on numeric columns.

30. **Write a PySpark code to find the top 3 highest salaries per department.**

```
from pyspark.sql.window import Window
from pyspark.sql.functions import row_number

df = spark.createDataFrame([
    (1, "HR", 3000),
    (2, "HR", 4000),
    (3, "Engineering", 5000),
    (4, "Engineering", 4500),
    (5, "Engineering", 6000)
], ["employee_id", "department", "salary"])

windowSpec = Window.partitionBy("department").orderBy(col("salary").desc())
top_3_df = df.withColumn("rank", row_number().over(windowSpec)).filter("rank <= 3")
top_3_df.show()
```

Explanation: We partition the data by department and order by salary in descending order. Then, row_number ranks rows within each partition, and we filter to keep only the top 3 ranks.

31. **Write code to add a column that shows the cumulative sum of salaries in each department.**

```
from pyspark.sql.functions import sum



windowSpec =
Window.partitionBy("department").orderBy("salary").rowsBetween(Window.unboundedPr
eceding, 0)

cum_sum_df = df.withColumn("cumulative_sum", sum("salary").over(windowSpec))

cum_sum_df.show()
```

Explanation: The cumulative_sum is calculated within each department by defining a window from the start to the current row (unboundedPreceding to currentRow) and applying sum over this window.

32. **How would you filter out rows with any null values?**

```
df = spark.createDataFrame([

    (1, "John", 1000),

    (2, None, 2000),

    (3, "Mike", None)

], ["id", "name", "salary"])



filtered_df = df.na.drop("any")

filtered_df.show()
```

Explanation: Using na.drop("any") removes rows that have at least one null value. This is useful for keeping only complete rows.

33. **Write a PySpark code to calculate the difference in salary between consecutive rows.**

```
from pyspark.sql.functions import lag



windowSpec = Window.orderBy("salary")

salary_diff_df = df.withColumn("salary_diff", col("salary") - lag("salary",
1).over(windowSpec))
```

```
salary_diff_df.show()
```

Explanation: We use lag to access the previous row's salary and calculate the difference by subtracting the lagged salary from the current one.

34. **Write a PySpark code to rename all columns by adding a prefix.**

```
new_df = df.toDF(*["new_" + c for c in df.columns])

new_df.show()
```

Explanation: This code simply renames each column by appending a prefix using toDF() to update the column names.

35. **How would you split a column of strings by a delimiter and expand it into multiple columns?**

```
from pyspark.sql.functions import split



df = spark.createDataFrame([("Alice,Bob,Charlie",)], ["names"])



expanded_df = df.select(split(df["names"], ",").alias("split_names"))

expanded_df.show(truncate=False)
```

Explanation: split() is used here to split a column by a delimiter (,) and create a new array column. Each item in the array can then be accessed individually.

36. **Write code to find employees who earn more than the department average.**

```
avg_salary_df = df.groupBy("department").agg(avg("salary").alias("avg_salary"))

joined_df = df.join(avg_salary_df, "department").filter(df["salary"] >
avg_salary_df["avg_salary"])

joined_df.show()
```

Explanation: We calculate the department average and join it with the original DataFrame, then filter for employees with salaries above the department average.

37. **Write a PySpark code to convert a DataFrame to Pandas and back to PySpark.**

```
pandas_df = df.toPandas()

spark_df = spark.createDataFrame(pandas_df)
```

```
spark_df.show()
```

Explanation: We convert a PySpark DataFrame to Pandas with toPandas(), then use createDataFrame() to convert it back to PySpark. This is useful for data manipulation requiring both APIs.

38. **Write code to add a row number column without partitioning.**

```
from pyspark.sql.functions import monotonically_increasing_id


df_with_row_num = df.withColumn("row_num", monotonically_increasing_id())

df_with_row_num.show()
```

Explanation: monotonically_increasing_id() adds a unique ID to each row without requiring partitioning.

39. **Write PySpark code to group data by year from a date column.**

```
from pyspark.sql.functions import year


df = spark.createDataFrame([("2022-01-01",)], ["date"])

year_df = df.groupBy(year("date").alias("year")).count()

year_df.show()
```

Explanation: year() extracts the year from the date column, allowing grouping by year.

40. **How would you write a custom UDF to capitalize a string column?**

```
from pyspark.sql.functions import udf

from pyspark.sql.types import StringType


def capitalize_string(s):

    return s.capitalize() if s else None


capitalize_udf = udf(capitalize_string, StringType())

df = df.withColumn("capitalized_name", capitalize_udf("name"))
```

```
df.show()
```

Explanation: We define a UDF that capitalizes strings, then apply it to the specified column.

41. **Write code to create a rolling average of a column over a window of 3 rows.**

```
from pyspark.sql.functions import avg



windowSpec = Window.orderBy("id").rowsBetween(-2, 0)

rolling_avg_df = df.withColumn("rolling_avg", avg("salary").over(windowSpec))

rolling_avg_df.show()
```

Explanation: We create a rolling window of 3 rows and calculate the average for each window.

42. **How would you drop all columns with more than 50% null values?**

```
threshold = int(0.5 * df.count())

filtered_df = df.dropna(thresh=threshold)

filtered_df.show()
```

Explanation: We set a threshold for the number of null values allowed and drop columns that exceed it.

43. **Write PySpark code to add a column that shows the day of the week from a date column.**

```
from pyspark.sql.functions import dayofweek



df = spark.createDataFrame([("2022-01-01",)], ["date"])

df = df.withColumn("day_of_week", dayofweek("date"))

df.show()
```

Explanation: dayofweek() extracts the day of the week, with 1 representing Sunday and so on.

44. **How would you add a suffix to all columns in a DataFrame?**

```
df_with_suffix = df.toDF(*[c + "_suffix" for c in df.columns])
```

```
df_with_suffix.show()
```

Explanation: Similar to adding a prefix, we use toDF() with modified column names to add a suffix.

45. **Write a code snippet to count the distinct values in each column of a DataFrame.**

```
distinct_counts = {col: df.select(col).distinct().count() for col in df.columns}

print(distinct_counts)
```

Explanation: We use a dictionary comprehension to count distinct values for each column and print the results.

46. **Write PySpark code to filter out rows with non-ASCII characters in a column.**

```
from pyspark.sql.functions import col



df_filtered = df.filter(col("name").rlike("^[\\x00-\\x7F]+$"))

df_filtered.show()
```

Explanation: Using rlike, we filter rows that match the ASCII character range (\\x00-\\x7F).

47. **How do you concatenate two DataFrames with the same schema in PySpark?**

```
concatenated_df = df1.union(df2)

concatenated_df.show()
```

Explanation: union appends rows from df2 to df1 when they have the same schema.

48. **Write code to calculate the percentage contribution of each row in a column.**

```
from pyspark.sql.functions import sum



total_sum = df.agg(sum("salary")).first()[0]

df_with_percentage = df.withColumn("percentage", (df["salary"] / total_sum) * 100)

df_with_percentage.show()
```

Explanation: We calculate the total sum of the column, then compute the percentage contribution of each row.

49. **Write code to convert a JSON string column to multiple columns.**

```
from pyspark.sql.functions import from_json

from pyspark.sql.types import StructType, StructField, StringType


schema = StructType([

    StructField("name", StringType(), True),

    StructField("age", StringType(), True)

])


df = df.withColumn("json_data", from_json("json_column", schema))

df = df.select("json_data.*")

df.show()
```

Explanation: from_json parses the JSON string into structured columns based on a specified schema.

50. **Write PySpark code to pivot a DataFrame with dynamic columns.**

```
pivoted_df = df.groupBy("id").pivot("category").agg(sum("value"))

pivoted_df.show()
```

Explanation: pivot creates a column for each unique value in category, aggregating values using sum.

Scenario Based Questions

1. "Suppose you have a DataFrame with duplicate records. How would you remove duplicates and keep only the first occurrence?"

- **Answer**: I would use dropDuplicates() to remove duplicates. By default, it retains the first occurrence, but if I need to specify certain columns, I can do so to make sure I'm dropping duplicates based on only those columns.

```
deduped_df = df.dropDuplicates(["column_name"])
```

```
deduped_df.show()
```

2. "How would you find the average salary by department from a large dataset using PySpark?"

- **Answer**: I'd group by the department column and then apply the avg function on the salary column. This way, we get the average salary per department in a distributed manner.

```
avg_salary_df = df.groupBy("department").agg(avg("salary").alias("avg_salary"))

avg_salary_df.show()
```

3. "You have a DataFrame of sales data and want to find the top 5 products based on sales volume. How would you do that?"

- **Answer**: I would group by product_id, sum up the sales_volume, and then order by this aggregated sum in descending order. After that, I'd limit the results to the top 5.

```
top_products_df = df.groupBy("product_id").agg(sum("sales_volume").alias("total_sales")) \

                .orderBy(col("total_sales").desc()).limit(5)

top_products_df.show()
```

4. "How would you calculate the cumulative sum of a column in PySpark for each department?"

- **Answer**: I'd use a Window function with sum() to compute the cumulative sum within each department. The window specification would include an ordering on the salary or another relevant column.

```
from pyspark.sql.window import Window

from pyspark.sql.functions import sum



windowSpec = Window.partitionBy("department").orderBy("salary").rowsBetween(Window.unboundedPreceding, 0)
```

```
cum_sum_df = df.withColumn("cumulative_sum", sum("salary").over(windowSpec))

cum_sum_df.show()
```

5. Interviewer: "Suppose you need to find employees who joined within the last six months. How would you approach this?"

- **Answer**: I would use the date_sub() function to subtract six months from the current date and then filter rows where the join_date is greater than or equal to this value.

```
from pyspark.sql.functions import current_date, date_sub



recent_joins_df = df.filter(df["join_date"] >= date_sub(current_date(), 180))

recent_joins_df.show()
```

6. "How would you count the number of null values in each column of a DataFrame?"

- **Answer**: I'd use a combination of isNull() and when() functions to count null values for each column, then apply count() to get the total for each column.

```
from pyspark.sql.functions import col, isnan, when, count



null_counts = df.select([count(when(col(c).isNull() | isnan(col(c)), c)).alias(c) for c in
df.columns])

null_counts.show()
```

7. "How would you handle a scenario where you need to join two DataFrames on multiple columns?"

- **Answer**: In PySpark, we can use multiple columns in the join condition. I'd specify the columns in the on parameter as a list of tuples.

```
joined_df = df1.join(df2, (df1["column1"] == df2["column1"]) & (df1["column2"] ==
df2["column2"]), "inner")
```

```
joined_df.show()
```

8. "How would you convert a JSON string column into multiple columns based on the JSON keys?"

- **Answer**: I'd use from_json with a specified schema to parse the JSON string and expand it into separate columns.

```
from pyspark.sql.functions import from_json

from pyspark.sql.types import StructType, StructField, StringType



schema = StructType([

    StructField("name", StringType(), True),

    StructField("age", StringType(), True)

])



df = df.withColumn("json_data", from_json("json_column", schema))

df = df.select("json_data.*")

df.show()
```

9. "Can you explain how you would calculate the difference in salary between consecutive rows for each department?"

- **Answer**: I'd use the lag function to access the previous row's salary and calculate the difference for each row by subtracting the lagged salary from the current salary.

```
from pyspark.sql.functions import lag



windowSpec = Window.partitionBy("department").orderBy("salary")

salary_diff_df = df.withColumn("salary_diff", col("salary") - lag("salary", 1).over(windowSpec))
```

```
salary_diff_df.show()
```

10. "How would you handle a scenario where you need to find the second highest salary in each department?"

- **Answer**: I'd use the dense_rank window function, partitioned by department and ordered by salary in descending order. Then, I'd filter to get the rows where the rank is 2.

```
from pyspark.sql.functions import dense_rank



windowSpec = Window.partitionBy("department").orderBy(col("salary").desc())

second_highest_salary_df = df.withColumn("rank",
dense_rank().over(windowSpec)).filter(col("rank") == 2)

second_highest_salary_df.show()
```

---

11. "Suppose you have a column with a comma-separated list of values. How would you split this into multiple columns?"

- **Answer**: I'd use the split function to create an array column, then access each element of the array as a separate column.

```
from pyspark.sql.functions import split



df_split = df.withColumn("split_col", split(df["comma_separated_column"], ","))

df_split = df_split.select(

    col("split_col").getItem(0).alias("col1"),

    col("split_col").getItem(1).alias("col2"),

    col("split_col").getItem(2).alias("col3")

)

df_split.show()
```

12. "How would you calculate the total sales per region and filter out regions with sales less than a specific threshold?"

- **Answer**: I'd group by region and sum up the sales. Then, I'd filter regions where the total sales exceed the threshold.

```
total_sales_df = df.groupBy("region").agg(sum("sales").alias("total_sales"))

filtered_sales_df = total_sales_df.filter(total_sales_df["total_sales"] > threshold)

filtered_sales_df.show()
```

13. "How would you add a running total column for each category in your DataFrame?"

- **Answer**: Using a Window function with sum, I'd calculate the running total within each category.

```
running_total_df = df.withColumn("running_total",
sum("sales").over(Window.partitionBy("category").orderBy("date").rowsBetween(Window.
unboundedPreceding, 0)))

running_total_df.show()
```

14. "You need to calculate the average time between two timestamps for each user. How would you do that?"

- **Answer**: I'd use lag to get the previous timestamp for each user and calculate the difference. Then, I'd compute the average of these differences.

```
from pyspark.sql.functions import lag, avg



windowSpec = Window.partitionBy("user_id").orderBy("timestamp")

df = df.withColumn("time_diff", col("timestamp") - lag("timestamp", 1).over(windowSpec))

avg_time_diff_df = df.groupBy("user_id").agg(avg("time_diff").alias("avg_time_diff"))

avg_time_diff_df.show()
```

15. "How would you find the nth highest salary in a DataFrame?"

- **Answer**: I'd use dense_rank or row_number in a window function to rank salaries, and then filter to get the nth highest.

```
from pyspark.sql.functions import dense_rank



n = 3  # For the 3rd highest salary

windowSpec = Window.orderBy(col("salary").desc())

nth_salary_df = df.withColumn("rank", dense_rank().over(windowSpec)).filter(col("rank") == n)

nth_salary_df.show()
```

16. Interviewer: "Can you show how to calculate the average session duration for each user?"

Answer: I would calculate the duration between login and logout timestamps for each session and then group by user to get the average session duration.

```
from pyspark.sql.functions import unix_timestamp



df = df.withColumn("session_duration", unix_timestamp("logout_time") - unix_timestamp("login_time"))

avg_session_duration_df = df.groupBy("user_id").agg(avg("session_duration").alias("avg_session_duration"))

avg_session_duration_df.show()
```

17. "Suppose you need to pivot a DataFrame based on a category column. How would you do this in PySpark?"

- **Answer**: I'd use the pivot() function, which creates a new column for each unique value in the specified column. For instance, if I wanted to pivot on category and sum up values for each pivot, I'd do it like this:

```
pivoted_df = df.groupBy("id").pivot("category").agg(sum("value"))
```

```
pivoted_df.show()
```

18. "How would you handle a scenario where you need to extract the year from a date column and group by it?"

- **Answer**: I'd use the year() function to extract the year and then group by this extracted year column.

```
from pyspark.sql.functions import year


yearly_df = df.groupBy(year("date_column").alias("year")).count()

yearly_df.show()
```

19. "You need to merge two DataFrames based on a primary key, keeping only records that exist in both DataFrames. How would you do this?"

- **Answer**: This is an inner join, where I join on the primary key and only keep the records that have matches in both DataFrames.

```
merged_df = df1.join(df2, on="primary_key", how="inner")

merged_df.show()
```

20. "Suppose you have a nested JSON structure in a column. How would you flatten it in PySpark?"

- **Answer**: I'd use the select method with dot notation to access each nested field and create a flattened DataFrame.

```
flattened_df = df.select("id", "json_column.field1", "json_column.field2")

flattened_df.show()
```

---

21. "You have a DataFrame with timestamp columns and want to calculate the time difference between two columns. How would you do this?"

- **Answer**: I'd use unix_timestamp() to convert the timestamps to seconds, and then find the difference between these values.

```
from pyspark.sql.functions import unix_timestamp


df = df.withColumn("time_diff", unix_timestamp("end_time") - unix_timestamp("start_time"))

df.show()
```

22. Interviewer: "How would you write code to count the unique values in each column of a DataFrame?"

- **Answer**: I'd use a dictionary comprehension to count distinct values for each column.

```
distinct_counts = {col: df.select(col).distinct().count() for col in df.columns}

print(distinct_counts)
```

23. "If you have a DataFrame with a date column, how would you filter the data for only the past year?"

- **Answer**: I'd use date_sub and current_date() to get the date one year ago and filter rows where the date is greater than or equal to this value.

```
from pyspark.sql.functions import current_date, date_sub


last_year_df = df.filter(df["date_column"] >= date_sub(current_date(), 365))

last_year_df.show()
```

24. Interviewer: "How would you handle duplicate rows but keep the latest record based on a timestamp column?"

- **Answer**: I'd use the row_number() window function partitioned by the unique columns and ordered by timestamp. Then, I'd filter to keep only the latest record (row number = 1).

```
from pyspark.sql.window import Window

from pyspark.sql.functions import row_number



windowSpec = Window.partitionBy("unique_id").orderBy(col("timestamp").desc())

deduped_df = df.withColumn("row_num",
row_number().over(windowSpec)).filter("row_num = 1").drop("row_num")

deduped_df.show()
```

---

25. "You need to extract the weekday from a date column and count records by weekday. How would you do this?"

- **Answer**: I'd use dayofweek to get the weekday from the date column and then group by this derived column to count occurrences.

```
from pyspark.sql.functions import dayofweek



weekday_df = df.withColumn("weekday",
dayofweek("date_column")).groupBy("weekday").count()

weekday_df.show()
```

26. Interviewer: "How would you calculate the rolling average over a window of 7 days in a time series data?"

- **Answer**: I'd define a window of 7 rows and use the avg function over this window to calculate the rolling average.

```
from pyspark.sql.functions import avg



windowSpec = Window.orderBy("date").rowsBetween(-6, 0)

rolling_avg_df = df.withColumn("rolling_avg", avg("value").over(windowSpec))

rolling_avg_df.show()
```

27. Interviewer: "If you had to compute the median salary by department, how would you approach this?"

- **Answer**: Since PySpark doesn't have a built-in median function, I'd use percentile_approx to approximate the median (50th percentile).

```
from pyspark.sql.functions import expr


median_salary_df = df.groupBy("department").agg(expr("percentile_approx(salary, 0.5)").alias("median_salary"))

median_salary_df.show()
```

28. "How would you use PySpark to filter out rows containing non-ASCII characters in a text column?"

- **Answer**: I'd use the rlike function with a regular expression to filter out rows containing only ASCII characters.

```
ascii_df = df.filter(col("text_column").rlike("^[\\x00-\\x7F]+$"))

ascii_df.show()
```

29. "If you need to add a prefix to all column names, how would you do it?"

- **Answer**: I'd use toDF() and modify the column names by adding a prefix.

```
prefixed_df = df.toDF(*[f"prefix_{c}" for c in df.columns])

prefixed_df.show()
```

---

30. "You have a DataFrame with sales data by month and want to calculate the month-over-month growth rate. How would you do that?"

- **Answer**: I'd use the lag function to access the previous month's sales and calculate the growth rate as the difference divided by the previous month's sales.

```
from pyspark.sql.functions import lag
```

```
windowSpec = Window.orderBy("month")

growth_df = df.withColumn("prev_sales", lag("sales").over(windowSpec))

growth_df = growth_df.withColumn("growth_rate", (col("sales") - col("prev_sales")) / col("prev_sales") * 100)

growth_df.show()
```

31. Interviewer: "Suppose you have an array column. How would you explode this column to create a new row for each element in the array?"

- **Answer**: I'd use the explode function to expand each array element into its own row.

```
from pyspark.sql.functions import explode



exploded_df = df.withColumn("exploded_column", explode("array_column"))

exploded_df.show()
```

32. Interviewer: "You need to find the distinct counts for each value in a specific column. How would you do that?"

- **Answer**: I'd use groupBy on the column and count() to get the distinct counts for each value.

```
distinct_count_df = df.groupBy("column_name").count()

distinct_count_df.show()
```

33. Interviewer: "If you want to find rows where a specific column contains a substring, how would you do that?"

- **Answer**: I'd use the contains function to filter rows based on the presence of the substring.

```
filtered_df = df.filter(df["column_name"].contains("substring"))

filtered_df.show()
```

34. Interviewer: "Suppose you have a dataset with employee information and department IDs. How would you perform a self-join to get manager-employee pairs based on a reporting hierarchy?"

- **Answer**: I'd use a join on the DataFrame with itself, matching employee ID with manager ID to get the hierarchy.

```
hierarchy_df = df.alias("a").join(df.alias("b"), col("a.manager_id") == col("b.employee_id"))

hierarchy_df.show()
```

35. Interviewer: "How would you convert a column containing comma-separated values into an array?"

- **Answer**: I'd use the split function to convert the comma-separated values into an array.

```
from pyspark.sql.functions import split



array_df = df.withColumn("array_column", split(col("comma_separated_column"), ","))

array_df.show()
```

36. "How would you calculate the difference between the highest and lowest values in a specific column within each group?"

- **Answer**: I'd use max() and min() functions within a groupBy clause to calculate the range for each group and then subtract the minimum from the maximum.

```
range_df = df.groupBy("group_column").agg((max("value_column") -
min("value_column")).alias("range"))

range_df.show()
```

37. Interviewer: "If you have a column with nested arrays, how would you flatten it in PySpark?"

- **Answer**: I'd use explode() in combination with explode_outer() if there are nested arrays, effectively unnesting the array structure into individual rows.

```
from pyspark.sql.functions import explode_outer


flattened_df = df.withColumn("flattened_column", explode_outer("nested_array_column"))

flattened_df.show()
```

38. "How would you add a suffix to all column names in a DataFrame?"

- **Answer**: I'd use toDF() to modify each column name by appending the suffix.

```
suffixed_df = df.toDF(*[f"{c}_suffix" for c in df.columns])

suffixed_df.show()
```

39. "How would you convert a PySpark DataFrame into a list of dictionaries, with each row represented as a dictionary?"

- **Answer**: I'd use collect() to gather the rows and then asDict() to convert each row into a dictionary.

```
rows = [row.asDict() for row in df.collect()]

print(rows)
```

40. "Suppose you need to filter rows where the values in a column fall within a specified range. How would you do this in PySpark?"

- **Answer**: I'd use filter() with the specified range conditions using between().

```
filtered_df = df.filter(df["column"].between(lower_bound, upper_bound))

filtered_df.show()
```

---

41. "How would you convert a PySpark DataFrame with numeric columns into a dense vector for each row for machine learning purposes?"

- **Answer**: I'd use the VectorAssembler class from pyspark.ml.feature.

```
from pyspark.ml.feature import VectorAssembler
```

```
assembler = VectorAssembler(inputCols=["column1", "column2"], outputCol="features")

vector_df = assembler.transform(df)

vector_df.show(truncate=False)
```

---

42. "How would you calculate the percentage of total sales for each product?"

- **Answer**: I'd calculate the total sales using agg() and then compute the percentage for each product by dividing the product sales by the total sales.

```
total_sales = df.agg(sum("sales").alias("total_sales")).collect()[0]["total_sales"]

percentage_df = df.withColumn("sales_percentage", (col("sales") / total_sales) * 100)

percentage_df.show()
```

---

43. "You have a column with JSON strings in it. How would you parse this JSON column into multiple columns based on keys?"

- **Answer**: I'd use from_json with a defined schema for the JSON structure.

```
from pyspark.sql.functions import from_json

from pyspark.sql.types import StructType, StructField, StringType


schema = StructType([StructField("key1", StringType(), True), StructField("key2",
StringType(), True)])

parsed_df = df.withColumn("json_data", from_json("json_column",
schema)).select("json_data.*")

parsed_df.show()
```

44. "If you need to select the top N records for each group, based on a specific column, how would you do that?"

- **Answer**: I'd use row_number() with a partition window to rank records within each group and then filter to get the top N.

```
from pyspark.sql.functions import row_number

from pyspark.sql.window import Window



windowSpec = Window.partitionBy("group_column").orderBy(col("value_column").desc())

top_n_df = df.withColumn("rank", row_number().over(windowSpec)).filter(col("rank") <= N)

top_n_df.show()
```

45. "How would you calculate a 7-day rolling average in PySpark?"

- **Answer**: I'd define a window of 7 days and apply the avg() function over this window.

```
rolling_window = Window.orderBy("date_column").rowsBetween(-6, 0)

rolling_avg_df = df.withColumn("7_day_rolling_avg",
avg("value_column").over(rolling_window))

rolling_avg_df.show()
```

---

46. "Suppose you have data with mixed upper and lower case values. How would you normalize them to lowercase?"

- **Answer**: I'd use the lower() function to convert all text to lowercase.

```
normalized_df = df.withColumn("normalized_column", lower("mixed_case_column"))

normalized_df.show()
```

47. "If you need to calculate the difference in values between consecutive rows, how would you do it?"

- **Answer**: I'd use lag() to get the previous row's value and subtract it from the current row's value.

```
windowSpec = Window.orderBy("date_column")
```

```
diff_df = df.withColumn("value_diff", col("value_column") - lag("value_column",
1).over(windowSpec))

diff_df.show()
```

48. "How would you generate a row number for each row, resetting at each group?"

- **Answer**: Using row_number() with a partitionBy specification would create a row number that resets for each group.

```
windowSpec = Window.partitionBy("group_column").orderBy("column")

row_num_df = df.withColumn("row_num", row_number().over(windowSpec))

row_num_df.show()
```

49. "If you need to find the most recent record for each user in a dataset, how would you approach this?"

- **Answer**: I'd use row_number() with a descending order on the timestamp column to keep only the latest record per user.

```
windowSpec = Window.partitionBy("user_id").orderBy(col("timestamp").desc())

latest_df = df.withColumn("rank", row_number().over(windowSpec)).filter("rank =
1").drop("rank")

latest_df.show()
```

50. "You have a DataFrame with a text column containing HTML tags. How would you remove the tags?"

- **Answer**: I'd use regular expressions in regexp_replace to remove HTML tags.

```
from pyspark.sql.functions import regexp_replace



clean_df = df.withColumn("clean_text", regexp_replace("html_column", "<[^>]+>", ""))

clean_df.show()
```

51. **"How would you count the number of times each value appears in a column?"**

- **Answer**: I'd use groupBy() on the column and apply count() to get the frequency of each value.

```
count_df = df.groupBy("column_name").count()

count_df.show()
```

52. **"How would you calculate the moving sum for each group in a column?"**

- **Answer**: I'd use a window function with sum() and partition it by the group column.

```
moving_sum_df = df.withColumn("moving_sum",
sum("value_column").over(Window.partitionBy("group_column").orderBy("date_column").
rowsBetween(Window.unboundedPreceding, 0)))

moving_sum_df.show()
```

53. "If you have a JSON file in HDFS and need to load it into PySpark, how would you do this?"

- **Answer**: I'd use spark.read.json() to load the JSON file from HDFS.

```
df = spark.read.json("hdfs://path/to/file.json")

df.show()
```

54. "How would you handle duplicate rows based on a specific column, keeping only the latest record?"

- **Answer**: Using row_number() with a partition by the column and an order by timestamp, I'd filter for the latest record.

```
deduped_df = df.withColumn("row_num",
row_number().over(Window.partitionBy("column").orderBy(col("timestamp").desc()))).filter
("row_num = 1").drop("row_num")

deduped_df.show()
```

55. Interviewer: "How would you concatenate two columns with a delimiter in between?"

- **Answer**: I'd use the concat_ws() function to join the columns with a delimiter.

```
from pyspark.sql.functions import concat_ws


concatenated_df = df.withColumn("full_column", concat_ws("-", "column1", "column2"))

concatenated_df.show()
```

---

56. Interviewer: "You need to sample 10% of rows from a DataFrame. How would you do this?"

- **Answer**: I'd use the sample() method with a fraction of 0.1 to get 10% of the rows.

```
sample_df = df.sample(0.1)

sample_df.show()
```

---

57. Interviewer: "How would you join two DataFrames and filter the result for matching records in both DataFrames?"

- **Answer**: I'd use an inner join to keep only matching records from both DataFrames.

```
joined_df = df1.join(df2, "join_column", "inner")

joined_df.show()
```

58. "If you want to pivot a DataFrame by two columns, how would you do that?"

- **Answer**: I'd use groupBy() with multiple columns, followed by pivot() on one of the columns.

```
pivoted_df = df.groupBy("column1", "column2").pivot("pivot_column").agg(sum("value"))

pivoted_df.show()
```

59. "How would you create a running average in each group by applying a moving window?"

- **Answer**: I'd use the avg() function over a moving window defined by each group.

```
moving_avg_df = df.withColumn("running_avg",
avg("value").over(Window.partitionBy("group_column").orderBy("date").rowsBetween(Window.unboundedPreceding, 0)))

moving_avg_df.show()
```

60. "If you need to sort by multiple columns, how would you do that in PySpark?"

- **Answer**: I'd use orderBy() with a list of columns and specify the order (ascending/descending) for each column.

```
sorted_df = df.orderBy(["column1", "column2"], ascending=[True, False])

sorted_df.show()
```

1. **Scenario: You have a dataset of customer transactions with columns customer_id, transaction_id, and amount. How would you find the total transaction amount for each customer?**

- **Solution**: I would group by customer_id and sum up the amount for each customer.

```
from pyspark.sql import SparkSession

from pyspark.sql.functions import sum



# Sample DataFrame

transactions_df = spark.createDataFrame([

    (1, "T1", 100),

    (1, "T2", 150),

    (2, "T3", 200),
```

```
    (3, "T4", 300),
], ["customer_id", "transaction_id", "amount"])


# Group by customer_id and sum up the amount
total_amount_df =
transactions_df.groupBy("customer_id").agg(sum("amount").alias("total_amount"))
total_amount_df.show()
```

2. Scenario: Given a dataset of employee records with columns employee_id, department, and salary, find the highest salary in each department.

- **Solution**: I would use groupBy on department and apply max on the salary column.

```
from pyspark.sql.functions import max


# Sample DataFrame
employees_df = spark.createDataFrame([
    (1, "HR", 4000),
    (2, "Engineering", 5000),
    (3, "HR", 4500),
    (4, "Engineering", 6000),
], ["employee_id", "department", "salary"])


# Group by department and find the highest salary
highest_salary_df =
employees_df.groupBy("department").agg(max("salary").alias("highest_salary"))
highest_salary_df.show()
```

3. Scenario: You have a dataset with columns user_id, activity, and timestamp. Find the last activity for each user.

- **Solution**: I would use row_number with a descending order on timestamp, and filter for the first row for each user_id.

```
from pyspark.sql.window import Window

from pyspark.sql.functions import row_number


# Sample DataFrame

activity_df = spark.createDataFrame([

   (1, "login", "2024-10-01 10:00:00"),

   (1, "purchase", "2024-10-01 12:00:00"),

   (2, "login", "2024-10-02 09:00:00"),

], ["user_id", "activity", "timestamp"])


# Define window and filter for last activity

windowSpec = Window.partitionBy("user_id").orderBy(activity_df["timestamp"].desc())

last_activity_df = activity_df.withColumn("rank",
row_number().over(windowSpec)).filter("rank = 1").drop("rank")

last_activity_df.show()
```

---

4. Scenario: In a sales dataset with region, product_id, and sales, find the total sales per product in each region.

- **Solution**: Group by region and product_id, then sum the sales column.

```
# Sample DataFrame

sales_df = spark.createDataFrame([

   ("East", "P1", 100),

   ("East", "P2", 150),
```

```
    ("West", "P1", 200),

    ("West", "P2", 250),

], ["region", "product_id", "sales"])



# Group by region and product_id to get total sales

total_sales_df = sales_df.groupBy("region", "product_id").agg(sum("sales").alias("total_sales"))

total_sales_df.show()
```

5. Scenario: For a weather dataset with columns date, city, and temperature, calculate the average temperature for each city.

- **Solution**: I would group by city and compute the average of temperature.

```
from pyspark.sql.functions import avg



# Sample DataFrame

weather_df = spark.createDataFrame([

    ("2024-10-01", "New York", 20),

    ("2024-10-01", "Los Angeles", 25),

    ("2024-10-02", "New York", 22),

], ["date", "city", "temperature"])



# Group by city and calculate average temperature

avg_temp_df = weather_df.groupBy("city").agg(avg("temperature").alias("avg_temperature"))

avg_temp_df.show()
```

6. Scenario: Given a dataset with columns student_id, subject, and score, find the highest score per subject.

- **Solution**: Group by subject and use max on score to find the highest score in each subject.

```
# Sample DataFrame
scores_df = spark.createDataFrame([
    (1, "Math", 90),
    (2, "Math", 95),
    (1, "Science", 85),
    (2, "Science", 88),
], ["student_id", "subject", "score"])

# Group by subject and find the maximum score
max_score_df = scores_df.groupBy("subject").agg(max("score").alias("max_score"))
max_score_df.show()
```

7. Scenario: For an inventory dataset with columns product_id, warehouse, and stock, find the total stock for each product across all warehouses.

- **Solution**: Group by product_id and use sum on stock.

```
# Sample DataFrame
inventory_df = spark.createDataFrame([
    ("P1", "WH1", 100),
    ("P1", "WH2", 150),
    ("P2", "WH1", 200),
], ["product_id", "warehouse", "stock"])

# Group by product_id and sum the stock
```

```
total_stock_df = inventory_df.groupBy("product_id").agg(sum("stock").alias("total_stock"))

total_stock_df.show()
```

---

8. Scenario: You have a dataset with customer_id, order_id, and order_date. Find the first order date for each customer.

- **Solution**: Use row_number with a window partitioned by customer_id and ordered by order_date.

```
from pyspark.sql.functions import row_number


# Sample DataFrame

orders_df = spark.createDataFrame([

    (1, "O1", "2024-01-01"),

    (1, "O2", "2024-01-05"),

    (2, "O3", "2024-02-01"),

], ["customer_id", "order_id", "order_date"])


# Define window and filter for first order date

windowSpec = Window.partitionBy("customer_id").orderBy("order_date")

first_order_df = orders_df.withColumn("rank",
row_number().over(windowSpec)).filter("rank = 1").drop("rank")

first_order_df.show()
```

9. Scenario: Given a dataset with columns city, date, and AQI, find the day with the highest AQI for each city.

- **Solution**: Use row_number with a window partitioned by city and ordered by AQI in descending order.

```
# Sample DataFrame
```

```
aqi_df = spark.createDataFrame([
    ("New York", "2024-10-01", 150),
    ("New York", "2024-10-02", 160),
    ("Los Angeles", "2024-10-01", 120),
], ["city", "date", "AQI"])


# Define window and get highest AQI day
windowSpec = Window.partitionBy("city").orderBy(aqi_df["AQI"].desc())
highest_aqi_df = aqi_df.withColumn("rank", row_number().over(windowSpec)).filter("rank = 1").drop("rank")
highest_aqi_df.show()
```

---

10. Scenario: In a dataset with columns user_id, session_id, and duration, calculate the total session duration for each user.

- **Solution**: Group by user_id and sum up duration to get the total session duration.

```
# Sample DataFrame
sessions_df = spark.createDataFrame([
    (1, "S1", 30),
    (1, "S2", 45),
    (2, "S3", 50),
], ["user_id", "session_id", "duration"])


# Group by user_id and calculate total duration
total_duration_df = sessions_df.groupBy("user_id").agg(sum("duration").alias("total_duration"))
total_duration_df.show()
```

11. Scenario: Given a dataset with columns customer_id, product_id, and purchase_amount, find the total purchase amount for each customer, and then rank customers based on their total purchase amount.

- **Solution**: First, group by customer_id and sum purchase_amount to get the total per customer. Then, use rank with a window to rank customers by their total purchase amount.

```
from pyspark.sql.functions import sum

from pyspark.sql.window import Window

from pyspark.sql.functions import rank


# Sample DataFrame

purchases_df = spark.createDataFrame([

    (1, "P1", 100),

    (1, "P2", 150),

    (2, "P3", 200),

    (3, "P4", 250),

], ["customer_id", "product_id", "purchase_amount"])


# Calculate total purchase amount per customer

total_purchase_df = purchases_df.groupBy("customer_id").agg(sum("purchase_amount").alias("total_purchase_amount"))


# Define window and rank customers by total purchase amount

windowSpec = Window.orderBy(total_purchase_df["total_purchase_amount"].desc())

ranked_df = total_purchase_df.withColumn("rank", rank().over(windowSpec))

ranked_df.show()
```

12. Scenario: In a dataset with columns order_id, customer_id, and order_date, find customers who placed more than one order on the same day.

- **Solution**: Use groupBy on both customer_id and order_date to count orders. Then filter for counts greater than 1.

```
from pyspark.sql.functions import count


# Sample DataFrame
orders_df = spark.createDataFrame([
    (1, "C1", "2024-10-01"),
    (2, "C1", "2024-10-01"),
    (3, "C2", "2024-10-02"),
], ["order_id", "customer_id", "order_date"])


# Group by customer_id and order_date, and count orders
multi_order_df = orders_df.groupBy("customer_id",
"order_date").agg(count("order_id").alias("order_count"))
multi_order_df = multi_order_df.filter(multi_order_df["order_count"] > 1)
multi_order_df.show()
```

13. Scenario: Given a sales dataset with columns product_id, sale_date, and amount, calculate the monthly sales for each product.

- **Solution**: I'd extract the month and year from sale_date, group by product_id, year, and month, and then sum the amount.

```
from pyspark.sql.functions import month, year


# Sample DataFrame
sales_df = spark.createDataFrame([
```

```
    ("P1", "2024-01-15", 100),

    ("P1", "2024-01-20", 200),

    ("P2", "2024-02-10", 300),

], ["product_id", "sale_date", "amount"])



# Extract year and month, group by product_id, year, month

monthly_sales_df = sales_df.withColumn("year", year("sale_date")).withColumn("month", month("sale_date"))

monthly_sales_df = monthly_sales_df.groupBy("product_id", "year", "month").agg(sum("amount").alias("monthly_sales"))

monthly_sales_df.show()
```

14. Scenario: You have a social media dataset with columns user_id, post_id, and timestamp. Find the number of posts made by each user within each month.

- **Solution**: Extract the month and year from the timestamp, then group by user_id, year, and month, and count the post_id.

```
# Sample DataFrame

social_media_df = spark.createDataFrame([

    (1, "P1", "2024-01-15 10:00:00"),

    (1, "P2", "2024-01-20 11:00:00"),

    (2, "P3", "2024-02-10 09:00:00"),

], ["user_id", "post_id", "timestamp"])



# Extract year and month, group by user_id, year, month

monthly_posts_df = social_media_df.withColumn("year", year("timestamp")).withColumn("month", month("timestamp"))
```

```
monthly_posts_df = monthly_posts_df.groupBy("user_id", "year",
"month").count().withColumnRenamed("count", "post_count")

monthly_posts_df.show()
```

15. Scenario: Given a dataset with city, temperature, and date, find the maximum temperature recorded in each city during each month.

- **Solution**: Extract the month and year from date, group by city, year, and month, and get the maximum temperature.

```
from pyspark.sql.functions import max


# Sample DataFrame

temp_df = spark.createDataFrame([

    ("New York", 20, "2024-01-10"),

    ("New York", 25, "2024-01-15"),

    ("Los Angeles", 22, "2024-02-20"),

], ["city", "temperature", "date"])


# Extract year and month, group by city, year, month, find max temperature

monthly_max_temp_df = temp_df.withColumn("year", year("date")).withColumn("month",
month("date"))

monthly_max_temp_df = monthly_max_temp_df.groupBy("city", "year",
"month").agg(max("temperature").alias("max_temperature"))

monthly_max_temp_df.show()
```

16. Scenario: Given a dataset of web sessions with columns session_id, user_id, and duration, calculate the average session duration for each user.

- **Solution**: Group by user_id and calculate the average duration.

```
from pyspark.sql.functions import avg
```

```
# Sample DataFrame

sessions_df = spark.createDataFrame([

    ("S1", 1, 10),

    ("S2", 1, 15),

    ("S3", 2, 20),

], ["session_id", "user_id", "duration"])


# Group by user_id and calculate average duration

avg_duration_df =
sessions_df.groupBy("user_id").agg(avg("duration").alias("avg_duration"))

avg_duration_df.show()
```

17. Scenario: In a sales dataset with columns transaction_id, customer_id, product_id, and amount, find the total sales per customer-product combination.

- **Solution**: Group by both customer_id and product_id, then sum the amount.

```
# Sample DataFrame

sales_df = spark.createDataFrame([

    ("T1", 1, "P1", 100),

    ("T2", 1, "P2", 150),

    ("T3", 2, "P1", 200),

], ["transaction_id", "customer_id", "product_id", "amount"])


# Group by customer_id and product_id, calculate total sales

customer_product_sales_df = sales_df.groupBy("customer_id",
"product_id").agg(sum("amount").alias("total_sales"))
```

```
customer_product_sales_df.show()
```

---

18. Scenario: Given a dataset of student grades with columns student_id, subject, and grade, calculate the average grade for each subject.

- **Solution**: Group by subject and calculate the average of grade.

```
# Sample DataFrame

grades_df = spark.createDataFrame([

    (1, "Math", 85),

    (2, "Math", 90),

    (1, "Science", 80),

    (2, "Science", 75),

], ["student_id", "subject", "grade"])


# Group by subject and calculate average grade

avg_grade_df = grades_df.groupBy("subject").agg(avg("grade").alias("avg_grade"))

avg_grade_df.show()
```

---

19. Scenario: For an employee dataset with columns employee_id, manager_id, and salary, find the total salary expense for each manager.

- **Solution**: Group by manager_id and sum the salary to find the total salary expense per manager.

```
# Sample DataFrame

employees_df = spark.createDataFrame([

    (1, 101, 5000),

    (2, 101, 6000),

    (3, 102, 7000),

], ["employee_id", "manager_id", "salary"])
```

```
# Group by manager_id and calculate total salary expense

total_salary_df =
employees_df.groupBy("manager_id").agg(sum("salary").alias("total_salary"))

total_salary_df.show()
```

20. Scenario: In a library dataset with columns user_id, book_id, and borrow_date, find the latest book borrowed by each user.

- **Solution**: Use row_number with a window partitioned by user_id and ordered by borrow_date in descending order, then filter for the first row for each user.

```
from pyspark.sql.functions import row_number


# Sample DataFrame

library_df = spark.createDataFrame([

   (1, "B1", "2024-01-01"),

   (1, "B2", "2024-01-15"),

   (2, "B3", "2024-02-10"),

], ["user_id", "book_id", "borrow_date"])


# Define window and filter for latest borrowed book

windowSpec = Window.partitionBy("user_id").orderBy(library_df["borrow_date"].desc())

latest_book_df = library_df.withColumn("rank",
row_number().over(windowSpec)).filter("rank = 1").drop("rank")

latest_book_df.show()
```

FREE Resources:

Pyspark Roadmap:

https://www.linkedin.com/posts/ajay026_pyspark-interview-questions-activity-7197455988197580800-4Gms?utm_source=share&utm_medium=member_desktop

Pyspark PDF:

https://www.linkedin.com/posts/ajay026_pyspark-guide-activity-7223223848408641536-OBfB?utm_source=share&utm_medium=member_desktop

My Pyspark Interview Experience:
https://www.linkedin.com/posts/ajay026_dataengineering-interviewquestions-pyspark-activity-7053260093546524672-_L9w?utm_source=share&utm_medium=member_desktop

4 Pyspark Projects for FREE:

https://www.linkedin.com/posts/ajay026_dataengineering-dataanalytics-dataanalysis-activity-7009002545285144576-9IIr?utm_source=share&utm_medium=member_desktop

Top 20 Pyspark Questions you should learn:

https://www.linkedin.com/posts/ajay026_pyspark-pysaprk-python-activity-7171353072491806720-zx4T?utm_source=share&utm_medium=member_desktop

Pyspark CheatSheet:

https://www.linkedin.com/posts/ajay026_dataengineering-machinelearnig-datascience-activity-7038876062130348032-PLpc?utm_source=share&utm_medium=member_desktop

Pyspark Scenario Based questions:
https://www.youtube.com/watch?v=bErn1bHAyqw&list=PL50mYnndduIF868zbDUPMBpJpwJwd4NZh